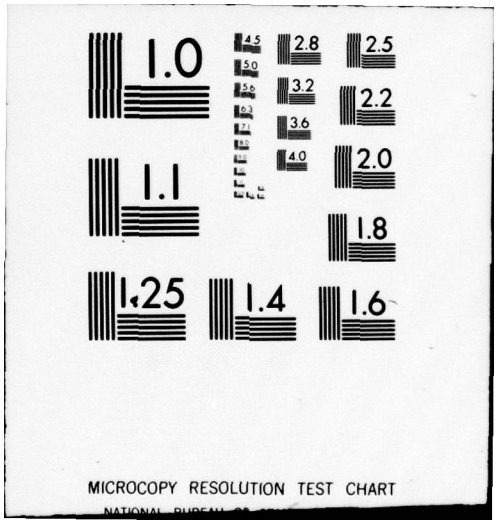
AD-A078 525 MARYLAND UNIV COLLEGE PARK F/G 12/1
THE FORMAL SPECIFICATION OF AN ADAPTIVE, PARALLEL FINITE ELEMEN--ETC(U)
DEC 78 P ZAVE N00014-77-C-0623
UNCLASSIFIED TR-715 NL

1 OF 1
ADA
078525

END
DATE FILMED
1 -80
DDC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3
7.1
MICROCOPY RESOLUTION TEST CHART

LEVEL II

Technical Report TR-715 December 1979
ONR-N00014-77-C-0623-715

(13)

(6)

THE FORMAL SPECIFICATION OF AN ADAPTIVE, PARALLEL FINITE ELEMENT System

(10) Pamela Zave

JUN 19 1979

Un of Maryland College Park

(12) 41

(9) Technical rept.

(11) Dec 78

(14) TR-715

*This research was supported in part by the Office of Naval Research, Mathematics Branch, under Contract N00014-77-C-0623.

Abstract

The FEARS system is an experimental finite element system which uses adaptivity and parallelism to alleviate computational problems of the method. This report contains a formal specification of the system which is complete down to, but not including, the numerical processing. It functions as a realistically complex example of the specification technique used.

THE FORMAL SPECIFICATION OF AN ADAPTIVE, PARALLEL FINITE ELEMENT SYSTEM

The finite element method is a numerical method with many applications and great practical importance. It also requires computations and data manipulations which are expensive and unwieldy. The FEARS (Finite Element, Adaptive Research Solver) project has attacked these problems on several fronts: Adaptive techniques have been developed to streamline the input phase, assist the user's intuition, and achieve an optimal cost/accuracy balance; parallelism has been introduced to provide data segmentation and the potential for exploitation of parallel hardware. The current status of this project is reported in [Zave & Rheinboldt 77] and [Zave & Rheinboldt 79].

The FEARS system has been designed as an experimental system to test the validity and usefulness of the ideas above. This technical report contains a formal specification of the current version of the system, which difers only slightly from that presented in [Zave & Rheinboldt 77] and [Zave & Rheinboldt 79].

The system is specified down to the level of, but not including, the numerical processing. The specification itself follows ideas presented in [Zave & Fitzwater 77], [Fitzwater & Zave 77], [Zave 78], and is published here as a realistically complex example of the application of those ideas.

I do not wish to duplicate the explanations of this specification technique which can be found in the referenced articles: This report is intended as a supplement to them. The syntax of the specification language used here, however, has been chosen rather arbitrarily (if the underlying concepts prove useful, the experience of many users will eventually contribute a widely acceptable syntax). For this reason, I will explain briefly the syntax used.

Specification of a System

The specification begins with a section of symbolic parameters, so that binding of their values can be delayed.

The system is specified as a set of processes. There is a separate section of the specification for each type of process, so that the system consists of (in order) bu + 1 "user" processes, one "control" process, b2 "two-subdomain" processes, b1 "one-subdomain" processes, b0 "zero-subdomain" processes, and bs "solver" processes.

A process is specified by its successor function. The first entry in the section specifying any process is a specification of its successor function. The successor function is a function on the set of all possible states of the process, called its "state space".

All sets used in the specification are specified (in alphabetical order) in the last section.

Specification of a Function

All functions are declared by naming their domains and ranges, as in:

```
function-name:
     "domain set expression"
---> "range set expression".
```

If the function is primitive then it is not formally specified further. Instead there will be a comment on its declaration describing what it should do.

If the function is non-primitive, it is defined by a function-expression in terms of "smaller" functions, arguments, and constants. A function-expression can use tuple-formation:

(x, y, z),

composition:

f(g(h)),

and selection:

[p1:f1, p2:f2, . . . pn:fn].

The selection evaluates to the first fi such that pi is true.

Since the specification language is "strongly typed", argument lists (formal and actual) must match the declared domains of their functions, and defining expressions must match

the declared ranges of their functions.

In the specification of each process, the successor function definition is followed by specifications of the functions used in it. They are followed by specifications of functions used in them, etc. For the most part the arrangement of functions is an intuitive linearization of the underlying tree structure. The definition f1 = (f3(f2), f5(f4)), for example, would be followed by specifications of f2, f3, f4, and f5 (in that order). Major subtrees, however, are separated into "groups". The root of a group subtree is reached via a statement such as:

turn: TURN GROUP

found in a place where a specification of the function "turn" would have been expected. "Turn" will then be found as the first specification in the "TURN GROUP" section, to be found a few pages later.

Specification of a Set

All sets are declared simply by giving their names. A primitive set carries a comment describing what it should contain. A non-primitive set is defined by a set-expression in terms of "smaller" sets and constants.

A set-expression can be made up of enumerations of constants:

{ true, false },

unions:

SET-ONE U SET-TWO,

and cross-products (cross-product has precedence over union):

SET-ONE x SET-TWO.

The state space of a process is defined implicitly by the set-expression giving the domain and range of its successor function. If an initial value is needed, it is given after the name of the set of which it is an element, as in:

```
successor-function:
        BOOLEAN . . true . . x OTHER-SET
---> BOOLEAN x OTHER-SET.
```

Intrinsic Primitives

The intrinsic primitive sets are the integers (I) and the real numbers (R).

There are a number of intrinsic primitive functions:

"proj-x-y", x and y integers, projects an x-tuple onto its first y components;

"equ" is a predicate which tests any two objects of the same type for equality;

"sum" and "max" perform the indicated operations on any number of numeric arguments;

"diff", "quotient", and "product" are binary arithmetic operators;

"gte" and "lte" are the binary numeric predicates "greater than or equal", "less than or equal";

"xc-t", t a string, is an exchange function of type XC and class t;

"xa-t", t a string, is an exchange function of type XA and class t.

Repetition

Any semantically meaningful expression, such as a function definition, an element of a tuple, an application of a function, a "pi:fi" in a selection, a constant in a set enumeration, a union term in a set-expression, or a cross-product term in a set-expression, can be the subject of indexed repetition. A simple, and fairly restrictive, syntax seems to be sufficient. To repeat a function specification, for instance, one would write:

```
j1<"function-name(j)"
        "domain-set-expression(j)"
   ---> "range-set-expression(j)"

   "function-name(j)" "parameter-list" =
        "function-expression(j)"
>8
```

This stands for eight specifications. Each quoted phrase stands for a string which may have "j" in it; if so, each instance of j will be replaced by one of the integers from 1 to 8 on that integer's repetition.

Any repetition construct can replace any single instance of the repeated expression. For instance, the expression

```
(a,b,j1<cj>3)
```

is equivalent to

```
(a,b,c1,c2,c3);
```

the expression

j1<f>3 (a,b)

is equivalent to

f(f(f(a,b)))

(f's domain and range must be the same, since it is applied twice to its own value); and the expression

j1< x SET-j-NAME >3

is equivalent to

SET-1-NAME x SET-2-NAME x SET-3-NAME.

The Specification

The remainder of this report is the specification itself.

PARAMETERS

BOUNDS

- bu bound on the number of users minus 1
- b2 bound on the number of two-subdomains
- b1 bound on the number of one-subdomains
- b0 bound on the number of zero-subdomains
- bs bound on the number of solvers
- bc bound on the number of commands per turn
- bp bound on the number of passes per command
- bt bound on the number of pairs in a two-subdomain report

CONSTANTS

- ch the highest possible error value
- cr the cost per element of refinement
- cm the cost of computing a micro-stiffness matrix
- ce the cost per element of computing error indicators
- cs the cost per point of setting up a linear system
- cl the cube of the cost per point of solving a linear system
- ct the cost per element of refining and solving for the new point

USER PROCESSES

SUCCESSOR GROUP

```
j><user-j-successor:
        PHASE . . first . . x FLAG x STATUS x
        ADJACENCY x MESH x POSTDATA
    ---> PHASE x FLAG x STATUS x ADJACENCY x MESH x POSTDATA

    user-j-successor(p,f,s,a,m,d)  =
        [equ(p,first):
             first-step(initial-j-adjacency)
        ,
         equ(p,other):
             other-step(turn(f,s,m),a,d)
        ]
>ɔu
```

```
    initial-j-adjacency:  DOMAIN INITIALIZATION GROUP

    first-step:
        ADJACENCY
    ---> PHASE x FLAG x STATUS x ADJACENCY x MESH x POSTDATA

    first-step(a)  =
        (other,
         initial-flag,
         initial-status,
         a,
         initial-mesh(a),
         initial-postdata
        )
```

```
    initial-flag:
    ---> FLAG
```

This non-deterministic function represents the user's choice of initial "manual override" flag.

```
    initial-status:
    ---> STATUS
```

This non-deterministic function represents the user's choice of initial status record.

```
    initial-mesh:
        ADJACENCY
    ---> MESH
```

This non-deterministic function represents the creation of all user-specific data. An initial mesh is based on the adjacency relation, which is the "skeleton" of the domain.

```
    initial-postdata:
    ---> POSTDATA
```

This non-deterministic function represents the user's choice of initial post-processing data.

```
turn:  TURN GROUP

other-step:
      (STATUS x MESH) x ADJACENCY x POSTDATA
---> PHASE x FLAG x STATUS x ADJACENCY x MESH x POSTDATA

other-step((s,m),a,d) =
      (other,
       new-flag(s,m),
       s,
       a,
       m,
       post-process(s,a,m,d)
      )


new-flag:
---> FLAG

      This non-deterministic function represents the user's
      choice of a new "manual override" flag between turns.


post-process:
      STATUS x ADJACENCY x MESH x POSTDATA
---> POSTDATA

      This non-deterministic function represents the individual
      user's post-processing, of any kind.  For instance, all
      numerical output is generated by this function.
           This function may include a sequence of queries to
      subdomain processes, for the purpose of getting DOMAIN
      information.  These queries take the forms:
           xc-r.2.j(xa-a.2.j((answer,q))), j in { 1, . . . ,
                b2 }, q in TWO-QUERY, yielding a result in
                TWO-ANSWER;
           xc-r.1.j(xa-a.1.j((answer,q))), j in { 1, . . . ,
                b1 }, q in ONE-QUERY, yielding a result in
                ONE-ANSWER;
           xc-r.0.j(xa-a.0.j((answer,q))), j in { 1, . . . ,
                b0 }, q in ZERO-QUERY, yielding a result in
                ZERO-ANSWER.
```

```
DOMAIN INITIALIZATION GROUP

j1<initial-j-adjacency:
   ---> ADJACENCY

   initial-j-adjacency =
        xc-u.j(null)
>bu


   initial-0-adjacency:
   ---> ADJACENCY

   initial-0-adjacency =
        initialize(initial-domain)


   initial-domain:
   ---> ADJACENCY x DOMAIN

        This non-deterministic function represents the creation
        of the domain for all users of the system.


   initialize:
        ADJACENCY x DOMAIN
   ---> ADJACENCY

   initialize(a,d) =
        proj-2-1
             (a
             ,
              (j1<initialize-j-user(a)>bu,
               initialize-control(a),
               j1<initialize-2.j-subdomain(d)>b2,
               j1<initialize-1.j-subdomain(d)>b1,
               j1<initialize-0.j-subdomain(d)>b0
              )
             )


j1<initialize-j-user:
        ADJACENCY
   ---> NULL

   initialize-j-user(a) =
        xc-u.j(a)
>bu


   initialize-control:
        ADJACENCY
   ---> NULL

   initialize-control(a) =
        xc-c(a)


j1<initialize-2.j-subdomain:
        DOMAIN
   ---> NULL

   initialize-2.j-subdomain(d) =
        xa-a.2.j((initialize-domain,domain-select(d,2.j)))
>b2


j1<initialize-1.j-subdomain:
```

```
        DOMAIN
    ---> NULL

    initialize-1.j-subdomain(d) =
        xa-a.1.j((initialize-domain,domain-select(d,1.j)))
>d1

j1<initialize-0.j-subdomain:
        DOMAIN
    ---> NULL

    initialize-0.j-subdomain(d) =
        xa-a.0.j((initialize-domain,domain-select(d,0.j)))
>d0

    domain-select:
        DOMAIN x ADDRESS
    ---> SUBDOMAIN

        This function selects the addressed subdomain segment.
```

```
TURN GROUP

    turn:
          FLAG x STATUS x MESH
    ---> STATUS x MESH

    turn(f,s,m) =
          receive-status-out
               (end-turn
                    (take-a-turn
                         (start-turn
                              (m,send-status-in(f,s)))))


    send-status-in:
          FLAG x STATUS
    ---> NULL

    send-status-in(f,s) =
          xa-c((f,s))


    start-turn:
          MESH x NULL
    ---> NULL

    start-turn(m,null) =
          proj-2-1
               (null
               ,(j1<start-2.j-subdomain(m)>b2,
                 j1<start-1.j-subdomain(m)>b1,
                 j1<start-0.j-subdomain(m)>b0
                )
               )


j1<start-2.j-subdomain:
          MESH
    ---> NULL

    start-2.j-subdomain(m) =
          xa-a.2.j((initialize-user,mesh-select(m,2.j)))
>b2


j1<start-1.j-subdomain:
          MESH
    ---> NULL

    start-1.j-subdomain(m) =
          xa-a.1.j((initialize-user,mesh-select(m,1.j)))
>b1


j1<start-0.j-subdomain:
          MESH
    ---> NULL

    start-0.j-subdomain(m) =
          xa-a.0.j((initialize-user,mesh-select(m,0.j)))
>b0


    mesh-select:
          MESH x ADDRESS
    ---> SUBMESH
```

This function selects the addressed subdomain segment.

```
take-a-turn:
     NULL
---> NULL

take-a-turn(null) =
     proj-2-1
          (null,j1<command>bc (go,null))


command:
     SIGNAL x REPORT
---> SIGNAL x REPORT

command(s,r) =
     [equ(s,stop):
          (stop,receive-report(send-command(stop)))
     ,
      equ(s,go):
          new-command(r)
     ]


new-command:
     REPORT
---> SIGNAL x REPORT

new-command(r) =
     evaluate
          (receive-report
               (send-command
                    (make-command(r))))


make-command:
     REPORT
---> COMMAND
```

This non-deterministic function represents the user's choice of his next command, based on the last report.

```
send-command:
     COMMAND
---> NULL

send-command(c) =
     xc-m(c)


receive-report:
     NULL
---> REPORT

receive-report(null) =
     xc-r(null)


evaluate:
     REPORT
---> SIGNAL x REPORT

evaluate(r) =
     [satisfied(r):
          (stop,null)
     ,
      true:
          (go,r)
     ]
```

```
satisfied:
     REPORT
---> { true, false }

     This non-deterministic predicate represents the user's
     choice whether or not to terminate his turn.

end-turn:
     NULL
---> MESH

end-turn(null) =
     unite-mesh
          (j1<end-2.j-subdomain>b2,
           j1<end-1.j-subdomain>b1,
           j1<end-0.j-subdomain>b0
          )


j1<end-2.j-subdomain:
---> TWO-SUBMESH

end-2.j-subdomain =
     xc-r.2.j(xa-a.2.j((finalize-user,null)))
>b2


j1<end-1.j-subdomain:
---> ONE-SUBMESH

end-1.j-subdomain =
     xc-r.1.j(xa-a.1.j((finalize-user,null)))
>b1


j1<end-0.j-subdomain:
---> ZERO-SUBMESH

end-0.j-subdomain =
     xc-r.0.j(xa-a.0.j((finalize-user,null)))
>b0


unite-mesh:
     j1< x TWO-SUBMESH >b2 x j1< x ONE-SUBMESH >b1 x
     j1< x ZERO-SUBMESH >b0
---> MESH

     This function joins segments to form a complete MESH
     structure again.


receive-status-out:
     MESH
---> STATUS x MESH

receive-status-out(m) =
     (xc-c(null),m)
```

CONTROL PROCESS

SUCCESSOR GROUP

```
control-successor:
     PHASE . . first . . x ADJACENCY
---> PHASE x ADJACENCY

control-successor(p,a) =
     [equ(p,first):
          (other, initial-control-adjacency)
     ,
      equ(p,other):
          proj-2-1((other,a)
                   ,
                   send-status-out
                        (cleanup-turn-status
                             (do-a-turn
                                  (a,receive-status-in)))
                   )


initial-control-adjacency:
---> ADJACENCY

initial-control-adjacency =
     xc-c(null)


receive-status-in:
---> FLAG x STATUS

receive-status-in =
     xc-c(null)


do-a-turn:  TURN GROUP


cleanup-turn-status:
     STATUS
---> STATUS

cleanup-turn-status((tcu,ctt,ctc,spc,ae,sn,r),s) =
     ((sum(tcu,ctt),0,ctc,spc,ae,sn,r),s)


send-status-out:
     STATUS
---> NULL

send-status-out(s) =
     xc-c(s)
```

```
TURN GROUP

    do-a-turn:
         ADJACENCY x (FLAG x STATUS)
    ---> STATUS

    do-a-turn(a,(f,s)) =
         turn-status
              (j1<obey-if-command>bc (a,f,s))

    obey-if-command:
         ADJACENCY x FLAG x STATUS
    ---> ADJACENCY x FLAG x STATUS

    obey-if-command(a,f,s) =
         stop-or-obey(a,f,s,receive-command)

    receive-command:
    ---> COMMAND

    receive-command =
         xc-m(null)

    stop-or-obey:
         ADJACENCY x FLAG x STATUS x COMMAND
    ---> ADJACENCY x FLAG x STATUS

    stop-or-obey(a,f,s,c) =
         (a,f,[equ(c,stop):
                   proj-2-4(s,send-report(null))
               ,
               true:
                   result(obey-command(a,f,s,c))
               ]
         )

    send-report:
         REPORT
    ---> NULL

    send-report(r) =
         xc-r(r)

    obey-command:
         ADJACENCY x FLAG x STATUS x COMMAND
    ---> STATUS

    obey-command(a,f,s,c) =
         command-status
              (j1<pass-if-needed>bb (gg,a,f,s,c,0))

    pass-if-needed:  PASS GROUP

    command-status:
         SIGNAL x ADJACENCY x FLAG x STATUS x COMMAND x STEP
    ---> STATUS

    command-status(g,a,f,s,c,t) =
         s
```

```
result:
      STATUS
---> STATUS

result(s) =
      proj-2-1
            (cleanup-command-status(s),send-report(report(s)))

cleanup-command-status:
      STATUS
---> STATUS

cleanup-command-status((tcu,ctt,ctc,spc,ae,sn,r),s) =
      ((tcu,sum(ctt,ctc),0,0,ae,sn,r),s)


report:
      STATUS
---> REPORT

report((tcu,ctt,ctc,spc,ae,sn,r),s) =
      (error-formula(ae,sn),ctc,spc,sum(ctt,ctc))


error-formula:
      ABSOLUTE-ERROR x SOLUTION-NORM
---> ERROR

      This function computes the relative error to be used as
      the measure of refinement.


turn-status:
      ADJACENCY x FLAG x STATUS
---> STATUS

turn-status(a,f,s) =
      s
```

PASS GROUP

```
pass-if-needed:
     SIGNAL x ADJACENCY x FLAG x STATUS x COMMAND x STEP
---> SIGNAL x ADJACENCY x FLAG x STATUS x COMMAND x STEP

pass-if-needed(g,a,f,s,c,t) =
     [equ(g,stop):
          (g,a,f,s,c,t)
     ,
      equ(g,slow):
          (stop,a,f,s,c,t)
     ,
      almost-finished(f,s,c):
          (slow,a,f,pass(a,s,long,(ch,0)),c,0)
     ,
      true:
          (go,
           a,
           f,
           pass(a,s,choose-length(c,t),cutoff(f,s)),
           c,
           mod-succ(c,t)
          )
     ]


almost-finished:
     FLAG x STATUS x COMMAND
---> { true, false }

almost-finished
     (f,
      ((tcu,ctt,ctc,spc,ae,sn,r),s),
      (ge,cl,spl,r)
     ) =
     [gte(spc,diff(spl,1)):  true,
      lte(error-formula(ae,sn),ge):  true,
      gte(ctc,cl):  true,
      gte(sum(ctc,long-pass-estimate(r,s)),cl):  true,
      true:  false
     ]


long-pass-estimate:
     RATIO x SUMMARIES
---> COST

     This formula estimates the cost of a long solution pass,
     based on summary data in the status record, the node/
     element ratio, and the constants cr,cm,ce,cs,cl.
          The original formula is (cr)(re) + (cm)(ae) +
     (ce)(ae) + (cs)(r)(ae) + (cl)[(r)(ae)]**3, where er is
     the estimated number of refined elements, and ae is the
     estimated number of active elements, based on
     SUMMARY-PAIR vectors.


choose-length:
     COMMAND x STEP
---> LENGTH

choose-length((ge,cl,spl,r),t) =
     [equ(r,t):  long
     ,
      true:      short
     ]
```

```
cutoff:
     FLAG x STATUS
---> CUTOFFS

cutoff(f,s) =
     [equ(f,yes):
          new-cutoffs
     ,
      true:
          (max-high-cutoff(s),max-low-cutoff(s))
     ]


new-cutoffs:
---> CUTOFFS

     This non-deterministic function represents the user's
     choice of new cutoffs, when the "manual override" flag
     is on.


max-high-cutoff:
     STATUS
---> CUTOFFS

max-high-cutoff(g,j1<(lae,lsn,lhc,llc,p)>b2) =
     max(j1<lhc>b2)


max-low-cutoff:
     STATUS
---> CUTOFFS

max-low-cutoff(g,j1<(lae,lsn,lhc,lhl,p)>b2) =
     max(j1<llc>b2)


pass:
     ADJACENCY x STATUS x LENGTH x CUTOFFS
---> STATUS

pass(a,s,l,c) =
     error-indication-phase(compute-phase(a,s,l,c))


compute-phase:
     ADJACENCY x STATUS x LENGTH x CUTOFFS
---> STATUS

compute-phase(a,s,l,c) =
     [equ(l,short):
          refine-and-short-solve(s,c)
     ,
      equ(l,long):
          long-solve(refine(a,s,c))
     ]


refine-and-short-solve:  REFINEMENT AND SHORT SOLUTION GROUP

refine:  REFINEMENT ONLY GROUP

long-solve:  LONG SOLUTION GROUP


error-indication-phase:
     STATUS
---> STATUS

error-indication-phase(s) =
```

```
        compute-errors(s,get-new-derivatives)

    get-new-derivatives:
    ---> j1< x NULL >b1

    get-new-derivatives =
        j1<new-1.j-derivatives>b1

j1<new-1.j-derivatives:
    ---> NULL

    new-1.j-derivatives =
        xc-r.1.j(xa-a.1.j((new-derivatives,null)))
>b1


    compute-errors:
        STATUS x j1< x NULL >b1
    ---> STATUS

    compute-errors(s,n) =
        newer-status(s,j1<error-2.j-subdomain>b2)

j1<error-2.j-subdomain:
    ---> SUMMARY

    error-2.j-subdomain =
        xc-r.2.j(xa-a.2.j((compute-errors,null)))
>b2


    newer-status:
        STATUS x j1< x SUMMARY >b2
    ---> STATUS

    newer-status
        (((tcu,ctt,ctc,spc,ae,sn,r),s),
          j1<(lae,lsn,lhc,llc,b)>b2
        ) =
        ((tcu,
          ctt,
          ctc,
          spc,
          sum(j1<lae>b2),
          sum(j1<lsn>b2),
          r
         ),
         j1<(lae,lsn,lhc,llc,b)>b2)
        )


    mod-succ:
        COMMAND x STEP
    ---> STEP

    mod-succ((ge,cl,spl,r),t) =
        [equ(r,t): 0
        ,
        true:      sum(t,1)
        ]
```

REFINEMENT AND SHORT SOLUTION GROUP

```
refine-and-short-solve:
      STATUS x CUTOFFS
---> STATUS

refine-and-short-solve(s,c) =
      short-new-status(s,refine-all-and-short-solve(c))


refine-all-and-short-solve:
      CUTOFFS
---> COUNT

refine-all-and-short-solve(c) =
      sum(j1<refine-2.j-and-short-solve(c)>b2)


j1<refine-2.j-and-short-solve:
      CUTOFFS
---> COUNT

refine-2.j-and-short-solve(c) =
      xc-r.2.j(xa-a.2.j((refine-and-short-solve,c)))
>b2


short-new-status:
      STATUS x COUNT
---> STATUS

short-new-status(((tcu,ctt,ctc,spc,ae,sn,r),s),c) =
      ((tcu,
        ctt,
        sum(ctc,short-pass-cost(c)),
        sum(spc,1),
        ae,
        sn,
        r
       ),
       s
      )


short-pass-cost:
      COUNT
---> COST

      This formula gives the cost of a short pass.  The
      original formula is (ct)(c).
```

```
REFINEMENT ONLY GROUP

    refine:
         ADJACENCY x STATUS x CUTOFFS
    ---> STATUS x SUBSET-INFO

    refine(a,s,c) =
         long-new-status
              (s,summarize-refinement(a,all-refine(c)))

    all-refine:
         CUTOFFS
    ---> REFINEMENT

    all-refine(c) =
         one-refine(two-refine(c))

    two-refine:
         CUTOFFS
    ---> TWO-REFINEMENT

    two-refine(c) =
         j1<refine-2.j-subdomain(c)>b2

j1<refine-2.j-subdomain:
         CUTOFFS
    ---> TWO-SUBREFINEMENT

    refine-2.j-subdomain(c) =
         xc-r.2.j(xa-a.2.j((refine,c)))
>b2

    one-refine:
         TWO-REFINEMENT
    ---> REFINEMENT

    one-refine(t) =
         (t,(j1<refine-1.j-subdomain>b1))

j1<refine-1.j-subdomain:
    ---> ONE-SUBREFINEMENT

    refine-1.j-subdomain =
         xc-r.1.j(xa-a.1.j((list-points,null)))
>b1

    summarize-refinement:
         ADJACENCY x REFINEMENT
    ---> COUNTS x SUBSET-INFO

    summarize-refinement(a,r) =
         (make-counts(r),find-subsets(a,r))

    make-counts:
         REFINEMENT
    ---> COUNTS

    make-counts((j1<(ap,ae,re)>b2),n) =
         (sum(j1<ae>b2),sum(j1<re>b2))
```

```
find-subsets:
     ADJACENCY x REFINEMENT
---> SOLVE-ORDERS x SOLVER-ADDRESSES
```

This primitive function creates the orders which will tell the two-subdomain and solver processes what to do during the solution phase. It works as follows:

A two-subdomain is "active" if its numbers of active points and active elements are non-zero. The solution subsets are the maximal sets of adjacent active two-subdomains (plus their adjacent one- and zero-subdomains).

The entry corresponding to a two-subdomain, a SOLVER-ADDRESS, is null if that two-subdomain is not active, and is the address of the solver process handling the subdomain's subset, if it is active.

There are as many active solver processes as there are subsets. The entries for all superfluous solver processes are null's.

For an active solver, the SOLVE-ORDER contains a list of all points in its subset belonging to one- and zero-subdomains, a count of all points in the subset, and a list of two-subdomains belonging to its subset.

```
long-new-status:
     STATUS x ( COUNTS x SUBSET-INFO )
---> STATUS x SUBSET-INFO

long-new-status(s,(c,b)) =
     (long-new-status-record(s,c,b),b)


long-new-status-record:
     STATUS x COUNTS x SUBSET-INFO
---> STATUS

long-new-status-record
     (((tcu,ctt,ctc,spc,ae,sn,r),s),
      (tae,tre),
      ((j1<(ol,pc,tsl)>bs),
       (j1<a>o2)
      )
     ) =
     ((tcu,
       ctt,
       sum(ctc,long-pass-cost(tae,tre,j1<pc>bs)),
       sum(spc,1),
       ae,
       sn,
       quotient(sum(j1<pc>bs),tae)
      ),
      s
     )


long-pass-cost:
     COUNT x COUNT x j1< x COUNT >bs
---> COST
```

This formula gives the cost of a long pass. The original formula is (cr)(tre) + (cm)(tae) + (ce)(tae) + j1< + (cs)(pc) >bs + j1< + (cl)(pc)**3 >bs.

```
LONG SOLUTION GROUP

    long-solve:
         STATUS x SUBSET-INFO
    ---> STATUS

    long-solve(s,(v,a)) =
         proj-2-1
              (s,
               (j1<order-2.j-subdomain(a)>b2,
                j1<order-s.j-solver(v)>bs
               )
              )

j1<order-2.j-subdomain:
         SOLVER-ADDRESSES
    ---> NULL

    order-2.j-subdomain(a) =
         xc-r.2.j(xa-a.2.j((partial-long-solution,
                                solver=select(a,2.j)
                           ))
                  )
>b2

    solver-select:
         SOLVER-ADDRESSES x ADDRESS
    ---> SOLVER-ADDRESS

         This function selects the addressed solver process
         address.

j1<order-s.j-solver:
         SOLVE-ORDERS
    ---> NULL

    order-s.j-solver(v) =
         xc-a.s.j(order-select(v,s.j))
>bs

    order-select:
         SOLVE-ORDERS x ADDRESS
    ---> SOLVE-ORDER

         This function selects the addressed solver process order.
```

TWO-SUBDOMAIN PROCESSES

```
j1<two-j-subdomain-successor:
        TWO-SUBDOMAIN x TWO-SUBMESH
    ---> TWO-SUBDOMAIN x TWO-SUBMESH

    two-j-subdomain-successor(d,m) =
        two-j-obey(d,m,get-2.j-order)
>b2

j1<get-2.j-order:
    ---> TWO-ORDER x TWO-DATA

    get-2.j-order =
        xc-a.2.j(null)
>b2

j1<two-j-obey:
        TWO-SUBDOMAIN x TWO-SUBMESH x (TWO-ORDER x TWO-DATA)
    ---> TWO-SUBDOMAIN x TWO-SUBMESH

    two-j-obey(d,m,(c,t)) =
        [equ(c,initialize-domain):
            (t,m)

        ,equ(c,initialize-user):
            (d,t)

        ,equ(c,finalize-user):
            proj-2-1((d,null),xc-r.2.j(m))

        ,equ(c,answer):
            proj-2-1((d,m),
                     xc-r.2.j(two-answer(d,m,t))

        ,equ(c,refine-and-short-solve):
            refinement-j-result(d,short-solve(d,m,t))

        ,equ(c,refine):
            refinement-j-result(d,(refine(d,m,t)))

        ,equ(c,partial-long-solution):
            partial-result(d,partial-long-solve(d,m,t),t)

        ,equ(c,complete-long-solution):
            long-solve-result(d,complete-long-solve(d,m,t))

        ,equ(c,compute-errors):
            error-j-result(d,errors(d,m))
        ]
>b2

    two-answer:
        TWO-SUBDOMAIN x TWO-SUBMESH x TWO-QUERY
    ---> TWO-ANSWER

        This function answers a query asked of a two-subdomain
        concerning its local data.

   *short-solve:
        TWO-SUBDOMAIN x TWO-SUBMESH x CUTOFFS
    ---> TWO-SUBMESH x COUNT

        This function refines the mesh structure according to the
```

```
        cutoffs, performs a short solution computation on the
        newly refined elements, and produces a count of refined
        elements.

   *refine:
        TWO-SUBDOMAIN x TWO-SUBMESH x CUTOFFS
    ---> TWO-SUBMESH x TWO-SUBREFINEMENT

        This function refines the mesh structure according to
        the cutoffs, and produces counts of active points, active
        elements, and refined elements.

j1<refinement-j-result:
        TWO-SUBDOMAIN x
        (TWO-SUBMESH x (COUNT U TWO-SUBREFINEMENT))
    ---> TWO-SUBDOMAIN x TWO-SUBMESH

    refinement-j-result(d,(m,r)) =
        proj-2-1((d,m),xc-r.2.j(r))
>j2

   *partial-long-solve:
        TWO-SUBDOMAIN x TWO-SUBMESH x SOLVER-ADDRESS
    ---> TWO-SUBDOMAIN x UPDATES

        This function does an internal partial solution (if
        this two-subdomain is active in this solution pass,
        which is true if the SOLVER-ADDRESS is not null),
        also producing updates for the solver process designated
        to solve this two-subdomain's subset.

    partial-result:
        TWO-SUBDOMAIN x (TWO-SUBMESH x UPDATES) x SOLVER-ADDRESS
    ---> TWO-SUBDOMAIN x TWO-SUBMESH

    partial-result(d,(m,u),a) =
        proj-2-1((d,m),
                 send-updates(u,a)
                )

    send-updates:
        UPDATES x SOLVER-ADDRESS
    ---> NULL

    send-updates(u,a) =
        [j1<equ(a,s.j):
                 xc-s.j(u)
        >bs]

   *complete-long-solve:
        TWO-SUBDOMAIN x TWO-SUBMESH x PARTIAL-SOLUTION
    ---> TWO-SUBMESH

        This function integrates the partial solution
        information, and completes the local solution.

    long-solve-result:
        TWO-SUBDOMAIN x TWO-SUBMESH
    ---> TWO-SUBDOMAIN x TWO-SUBMESH

    long-solve-result(d,m) =
        proj-2-1((d,m),xc-r.2.j(null))

   *errors:
```

```
        TWO-SUBDOMAIN x TWO-SUBMESH
    ---> TWO-SUBMESH x SUMMARY

        This function computes error indicators and prepares a
        summary.

j1<error-j-result:
        TWO-SUBDOMAIN x (TWO-SUBMESH x SUMMARY)
    ---> TWO-SUBDOMAIN x TWO-SUBMESH

    error-j-result(a,(m,s)) =
        proj-2-1((a,m),xc-r.2.j(s))
>b2
```

---

*This function may include a sequence of queries and updates to neighboring one- and zero- subdomain processes.

The queries take the forms:

xc-r.1.j(xa-a.1.j((answer,q))), j in {1, . . . b1}, q in ONE-QUERY, yielding a result in ONE-ANSWER;

xc-r.0.j(xa-a.0.j((answer,q))), j in {1, . . . b0}, q in ZERO-QUERY, yielding a result in ZERO-ANSWER.

The updates take the form:

xa-a.1.j((update,u)), j in {1, . . b1}, u in ONE-UPDATE.

ONE-SUBDOMAIN PROCESSES

```
j1<one-j-subdomain-successor:
        ONE-SUBDOMAIN x ONE-SUBMESH
    ---> ONE-SUBDOMAIN x ONE-SUBMESH

    one-j-subdomain-successor(d,m) =
        one-j-obey(d,m,get-one-j-order)
>b1

j1<get-one-j-order:
    ---> ONE-ORDER x ONE-DATA

    get-one-j-order =
        xc-a.1.j(null)
>b1

j1<one-j-obey:
        ONE-SUBDOMAIN x ONE-SUBMESH x (ONE-ORDER x ONE-DATA)
    ---> ONE-SUBDOMAIN x ONE-SUBMESH

    one-j-obey(d,m,(c,t)) =
        [equ(c,initialize-domain):
            (t,m)

        ,equ(c,initialize-user):
            (d,t)

        ,equ(c,finalize-user):
            proj-2-1((d,null),xc-r.1.j(m))

        ,equ(c,answer):
            proj-2-1((d,m),
                     xc-r.1.j(one-answer(d,m,t))

        ,equ(c,update):
            (d,one-update(m,t))

        ,equ(c,list-points):
            proj-2-1((d,m),
                     xc-r.1.j(list-points(m))
                    )

        ,equ(c,put-solutions):
            (d,put-solutions(m,t))

        ,equ(c,new-derivatives):
            derivatives-j-result(d,derivatives(m))
        ]
>b1

    one-answer:
        ONE-SUBDOMAIN x ONE-SUBMESH x ONE-QUERY
    ---> ONE-ANSWER

        This function answers a query asked of a one-subdomain
        concerning its local data.

    one-update:
        ONE-SUBMESH x ONE-UPDATE
    ---> ONE-SUBMESH

        This function performs an update to a one-subdomain's
        local information.
```

```
   list-points:
        ONE-SUBMESH
   ---> ONE-SUBREFINEMENT

        This function makes a list of all the points in the
        current one-submesh.


   put-solutions:
        ONE-SUBMESH x ONE-SOLUTIONS
   ---> ONE-SUBMESH

        This function updates the current one-submesh with new
        solutions.


   derivatives:
        ONE-SUBMESH
   ---> ONE-SUBMESH

        This function gets new directional derivatives.  It will
        make queries to neighboring two-subdomain processes, in
        the form:
             xc-r.2.j(xa-a.2.j((answer,q))), j in {1, . . . b2},
                  q in TWO-QUERY, yielding a result in
                  TWO-ANSWER.


j1<derivatives-j-result:
        TWO-SUBDOMAIN x TWO-SUBMESH
   ---> TWO-SUBDOMAIN x TWO-SUBMESH

   derivatives-j-result(d,m) =
        proj-2-1((d,m),xc-r.1.j(null))
>j1
```

ZERO-SUBDOMAIN PROCESSES

```
j1<zero-j-subdomain-successor:
        ZERO-SUBDOMAIN x ZERO-SUBMESH
    ---> ZERO-SUBDOMAIN x ZERO-SUBMESH

    zero-j-subdomain-successor(d,m) =
        zero-j-obey(d,m,get-zero-j-order)
>oJ

j1<get-zero-j-order:
    ---> ZERO-ORDER x ZERO-DATA

    get-zero-j-order =
        xc-a.J.j(null)
>oJ

j1<zero-j-obey:
        ZERO-SUBDOMAIN x ZERO-SUBMESH x (ZERO-ORDER x ZERO-DATA)
    ---> ZERO-SUBDOMAIN x ZERO-SUBMESH

    zero-j-obey(d,m,(c,t)) =
        [equ(c,initialize-domain):
            (t,m)

        ,equ(c,initialize-user):
            (d,t)

        ,equ(c,finalize-user):
            proj-2-1((d,null),xc-r.0.j(m))

        ,equ(c,answer):
            proj-2-1((d,m),
                     xc-r.0.j(zero-answer(d,m,t))

        ,equ(c,put-solution):
            (d,put-solution(m,t))
        ]
>oJ


    zero-answer:
        ZERO-SUBDOMAIN x ZERO-SUBMESH x ZERO-QUERY
    ---> ZERO-ANSWER

        This function answers a query asked of a zero-subdomain
        concerning its local data.


    put-solution:
        ZERO-SUBMESH x ZERO-SOLUTION
    ---> ZERO-SUBMESH

        This function updates the current zero-submesh with
        a new solution.
```

SOLVER PROCESSES

```
j1<solver-j-successor:
   ---> NULL

   solver-j-successor =
        send-solutions
             (solve
                  (update-j-system
                       (setup-j-system)))
>ds

j1<setup-j-system:
   ---> LINSYS

   setup-j-system =
        setup(xc-a.s.j(null))
>ds


   setup:
        SOLVE-ORDER
   ---> LINSYS x TWO-SUBDOMAIN-COUNT

        This function sets up a linear system solution from
        the information in the SOLVE-ORDER.  It also produces
        a count of the number of two-subdomains in this subset.


j1<update-j-system:
        LINSYS x TWO-SUBDOMAIN-COUNT
   ---> LINSYS

   update-j-system(s,c) =
        [k1<equ(c,k):
             l1<receive-j-and-update(s)>k
         >b2
        ]
>ds

j1<receive-j-and-update:
        LINSYS
   ---> LINSYS

   receive-j-and-update(l) =
        update(l,receive-j-updates)
>ds


j1<receive-j-updates:
   ---> UPDATES

   receive-j-updates =
        xc-s.j(null)
>ds


   update:
        LINSYS x UPDATES
   ---> LINSYS

        This function incorporates the updates from a
        two-subdomain.


   solve:
```

```
        LINSYS
    ---> SOLUTIONS

        This function solves the linear system.


    send-solutions:
        SOLUTIONS
    ---> NULL

    send-solutions(s) =
        proj-2-1
            (null,
             (j1<send-2.j-partial-solutions(s)>b2,
              j1<send-1.j-solutions(s)>b1,
              j1<send-0.j-solutions(s)>b0
             )
            )


j1<send-2.j-partial-solutions:
        SOLUTIONS
    ---> NULL

    send-2.j-partial-solutions(s) =
        [are-solutions-for(s,2.j):
            xa-a.2.j((complete-long-solution,
                      select-solutions(s,2.j)
                     ))
        ,
        true:null
        ]
>b2


j1<send-1.j-solutions:
        SOLUTIONS
    ---> NULL

    send-1.j-solutions(s) =
        [are-solutions-for(s,1.j):
            xa-a.1.j((put-solutions,select-solutions(s,1.j)))
        ,
        true:null
        ]
>b1


j1<send-0.j-solutions:
        SOLUTIONS
    ---> NULL

    send-0.j-solutions(s) =
        [are-solutions-for(s,0.j):
            xa-a.0.j((put-solution,select-solutions(s,0.j)))
        ,
        true:null
        ]
>b0


    are-solutions-for:
        SOLUTIONS x ADDRESS
    ---> { true, false }

        This predicate tells whether or nor the SOLUTIONS
        structure contains solutions (or partial solutions)
        for the addressed subdomain.


    select-solutions:
        SOLUTIONS x ADDRESS
```

---> PARTIAL-SOLUTION U ONE-SOLUTIONS U ZERO-SOLUTION

This function selects the addressed solution structure.

SETS

(In Alphabetical Order)

sets equal to the Reals and Integers
are defined where used as (R) and (I) respectively

ADDRESS =

{ j1<2.j>b2, j1<1.j>b1, j1<0.j>b0, j1<s.j>bs }

ADJACENCY

This data structure contains the complete adjacency relation on all two- , one- , and zero-subdomains.

COMMAND =

{ stop } U
GOAL-ERROR (R) x
COST-LIMIT (R) x
SOLUTION-PASS-LIMIT (I) x
RHYTHM (I)

COST (R)

COUNT (I)

COUNTS =

TOTAL-ACTIVE-ELEMENTS (I) x
TOTAL-REFINED-ELEMENTS (I)

CUTOFFS =

HIGH-CUTOFF (R) x
LOW-CUTOFF (R)

DOMAIN

This data structure contains mappings and bilinear forms. It must be segmented by subdomains, with each subdomain's segment containing (redundantly) the local adjacency relation.

ERROR (R)

FLAG =

{ yes, no }

LENGTH =

{ short, long }

MESH

This data structure contains a mesh with the most recent solutions attached, the user's boundary conditions, and the load functional. It must be segmented by subdomains.

NULL =

{ null }

ONE-ANSWER

This data structure contains an answer to a query sent to a one-subdomain process.

ONE-DATA =

ONE-SUBDOMAIN U ONE-SUBMESH U ONE-QUERY U ONE-UPDATE U NULL U ONE-SOLUTIONS

ONE-ORDER =

{ initialize-domain, initialize-user, finalize-user, answer, update, list-points, put-solutions, new-derivatives }

ONE-QUERY

This data structure contains a query sent to a one-subdomain process.

ONE-REFINEMENT =

j1< x ONE-SUBREFINEMENT >o1

ONE-SOLUTIONS

This data structure contains new solutions to be incorporated into a ONE-SUBMESH.

ONE-SUBDOMAIN

This data structure contains a segment of the DOMAIN data structure belonging to a one-subdomain.

ONE-SUBMESH

This data structure contains a segment of the MESH data structure belonging to a one-subdomain.

ONE-SUBREFINEMENT

This data structure contains a list of all the points in a one-subdomain, reported immediately after refinement. Since it will become part of a larger indexed data structure, it should be implemented as a fixed-length data structure.

ONE-UPDATE

This data structure contains the information for an

update to a one-subdomain process.

PARTIAL-SOLUTION

This data structure contains the partial solution information that a solver process sends back to a two-subdomain process belonging to the solver's subset.

PHASE =

{ first, other }

POSTDATA

This data structure contains the user's post-processing data, of any kind.

RATIO (R)

REFINEMENT =

TWO-REFINEMENT x
ONE-REFINEMENT

REPORT =

NULL U
ACHIEVED-ERROR (R) x
COST-USED (R) x
SOLUTION-PASS-NUMBER (I) x
TURN-COST (R)

SIGNAL =

{ go, slow, stop }

SOLUTIONS

This data structure contains the results of solving a linear system in a solver process.

SOLVE-ORDER =

NULL U
POINT-LIST x
POINT-COUNT (I) x
TWO-SUBDOMAIN-LIST

POINT-LIST

This data structure is a list of points to be solved for by a solver process.

TWO-SUBDOMAIN-LIST

This data structure is a list of two-subdomains in a subset.

SOLVE-ORDERS =

```
      j1< x SOLVE-ORDER >bs

SOLVER-ADDRESS =

      NULL U { j1<s.j>bs }

SOLVER-ADDRESSES =

      j1< x SOLVER-ADDRESS >b2

STATUS =

      GLOBAL-STATUS x
      SUMMARIES

      GLOBAL-STATUS =

            TOTAL-COST-USED (R) x
            COST-THIS-TURN (R) x
            COST-THIS-COMMAND (R) x
            SOLUTION-PASS-COUNT (I) x
            ABSOLUTE-ERROR (R) x
            SOLUTION-NORM (R) x
            RATIO

      SUMMARIES =

            j1< x SUMMARY >b2

STEP (I)

SUBDOMAIN =

      TWO-SUBDOMAIN U ONE-SUBDOMAIN U ZERO-SUBDOMAIN

SUBMESH =

      TWO-SUBMESH U ONE-SUBMESH U ZERO-SUBMESH

SUBSET-INFO =

      SOLVE-ORDERS x SOLVER-ADDRESSES

SUMMARY =

      LOCAL-ABSOLUTE-ERROR (R) x
      LOCAL-SOLUTION-NORM (R) x
      LOCAL-HIGH-CUTOFF (R) x
      LOCAL-LOW-CUTOFF (R) x
      j1< x SUMMARY-PAIR >bt

      SUMMARY-PAIR =

            ERROR-LEVEL (R) x
            ELEMENT-COUNT (I)

TWO-ANSWER

      This data structure contains an answer to a query sent to a
      two-subdomain process.
```

TWO-DATA =

TWO-SUBDOMAIN U TWO-SUBMESH U NULL U TWO-QUERY U CUTOFFS U SOLVER-ADDRESS U PARTIAL-SOLUTION

TWO-ORDER =

{ initialize-domain, initialize-user, finalize-user, answer, refine-and-short-solve, refine, partial-long-solution, complete-long-solution, compute-errors }

TWO-QUERY

This data structure contains a query sent to a two-subdomain process.

TWO-REFINEMENT =

j1< x TWO-SUBREFINEMENT >b2

TWO-SUBDOMAIN

This data structure contains a segment of the DOMAIN data structure belonging to a two-subdomain.

TWO-SUBDOMAIN-COUNT (I)

TWO-SUBMESH

This data structure contains a segment of the MESH data structure belonging to a two-subdomain.

TWO-SUBREFINEMENT =

ACTIVE-POINTS (I) x
ACTIVE-ELEMENTS (I) x
REFINED-ELEMENTS (I)

UPDATES

This data structure contains the updates that a two-subdomain process sends to the solver process solving the subset to which the two-subdomain belongs.

ZERO-ANSWER

This data structure contains an answer to a query sent to a zero-subdomain process.

ZERO-DATA =

ZERO-SUBDOMAIN U ZERO-SUBMESH U ZERO-QUERY U ZERO-SOLUTION

ZERO-ORDER =

{ initialize-domain, initialize-user, finalize-user, answer, quit-solution }

ZERO-QUERY

This data structure contains a query sent to a zero-subdomain process.

ZERO-SOLUTION (R)

ZERO-SUBDOMAIN

This data structure contains a segment of the DOMAIN data structure belonging to a zero-subdomain.

ZERO-SUBMESH

This data structure contains a segment of the MESH data structure belonging to a zero-subdomain.

References

[Fitzwater & Zave 77]
Fitzwater, D. R., and Zave, Pamela. "The Use of Formal Asynchronous Process Specifications in a System Development Process." Proceedings of the Sixth Texas Conference on Operating Systems, 1977, pp. 2B-21 - 2B-30.

[Zave 78]
Zave, Pamela. "Functional Specification of Asynchronous Processes and its Application to Requirements." Submitted for Publication, 1978.

[Zave & Fitzwater 77]
Zave, Pamela, and Fitzwater, D. R. "Specification of Asynchronous Interactions Using Primitive Functions." Univ. of Md. Comp. Sci. Dept. TR-598, 1977.

[Zave & Rheinboldt 77]
Zave, Pamela, and Rheinboldt, Werner C. Design of an Adaptive, Parallel Finite Element System." Univ. of Md. Comp. Sci. Dept. TR-593, 1977.

[Zave & Rheinboldt 79]
Zave, Pamela, and Rheinboldt, Werner C. "Design of an Adaptive, Parallel Finite Element System." To appear in Trans on Mathematical Software, 1979.